ANTWERP PAPERS IN LINGUISTICS

1975, nr. 1.

PARSING SYSTEMS FOR REGULAR AND CONTEXT-FREE LANGUAGES

Luc Steels

UNIVERSITEIT ANTWERPEN

UNIVERSITAIRE INSTELLING ANTWERPEN

Departementen Ger. en Rom.. Afdeling Linguïstiek

UNIVERSITEITSPLEIN 1 — 2610 WILRIJK -- TEL. 031.28.25.28

ABSTRACT

In this paper we approach the following problem: Given an arbitrary language describing device (e.g. a grammar), construct an algorithm which computes structural information for an arbitrary string of the language described by this device. Algorithms performing this task are called parsers and the problem itself is known as the 'recognition' or 'parsing' problem.

In the first part three devices accepting the set of regular or type 3 languages are presented: transition networks, finite state machines and regular grammars. Then parsers are constructed with these systems as data.

Basically a parser works on n-tuples (called tasks) containing all sorts of information, e.g. what symbol should be read in the inputstring, which rule can be applied, etc... .
Starting from a given initial task, new tasks are constructed from previous tasks by means of a recursive function. The execution of the function involves scanning the inputstring and consulting the grammar. After a finite number of steps, no more tasks can be created and from the set of tasks produced during the computation structural information can be filtered out.

In the second part of the paper three devices accepting the set of context-free or type 2 languages are presented: recursive transition networks, pushdown automata and context-free grammars. Then parsers are constructed for these systems with the same basic strategy.

Emphasis is laid on the construction of a fundamental theoretical framework rather than the description of sophisticated parsers and their implementations.

This paper is an attempt to create a theoretical framework for parsers. Parsers are systems taking as data grammars or other language describing devices and computing structural information for arbitrary strings of the described language.

It is clear that structural descriptions are of great importance . It is strange therefore that there is no interest within theoretical linguistics in finding exact methods to recognize structural information for a given inputstring according to a given grammar.

Although we approach this subject from a formal point of view, the main ideas and the final aim of the study arose from research in natural language processing. We personally think that only a careful (and therefore formal) study of the models underlying the implementations will lead to sound results.

Due to time and space limits we cannot but sketch a framework. It is possible to build more efficient (and therefore more complicated) parsers, but we must start somewhere.

The paper has two parts:
(i) type 3 parsers: i.e. parsingsystems for type 3 grammars, finite automata or transition networks.
(ii) type 2 parsers: i.e. parsing systems for context-free grammars, pushdown automata and recursive transition networks.

In natural language processing type 3 grammars can be used for the construction and consultation of the lexicon and for orthographemic rewriting. Type 2 grammars can be used for morphological and syntactic analysis. Together both types of systems lead to a complete automatic analysis of the surface structure of a natural language. Referencesabout applications are given in 2.7..

We included some problems which are solved at the end (in appendix A.) to help the reader in understanding the text. The introduction and emphasis on transition networks must be seen in the light of the growing popularity of this form of representation especially in computational linguistics.

Many ideas expressed in this paper are influenced by the seminars at the I.S. for Mathematical and Computational Linguistics in Pisa by M. Kay and W. Woods. The background for this study was provided by the exciting seminars in formal systems by prof. dr. G. Rozenberg. I thank the reading committee of the Antwerp papers in linguistics who accepted this paper for publication and especially prof. dr. R.G. Van de Velde who took the pain to examine with great care the manuscripts and by whose remarks the readibility of the text could be highly improved.
Of course the author is fully responsible for all remaining errors.

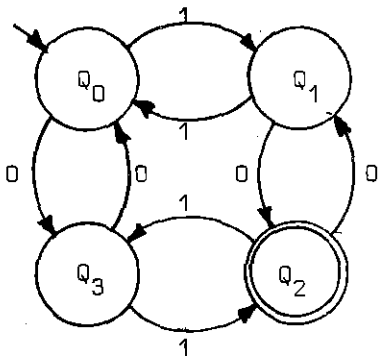
Contents

1. Simple transition networks and related systems.

1. TRANSITION NETWORKS

Definition 1.1. A simple transition network (for short t.n.) is a set of states represented by circles and a set of transitions from one state to another represented by directed labeled lines between the circles. The labels are the conditions for a transition to take place.
The initial or start state is marked by a little arrow. Final states are indicated by a double circle.

Example 1.1. The following diagram D is a particular instance of a t.n.:



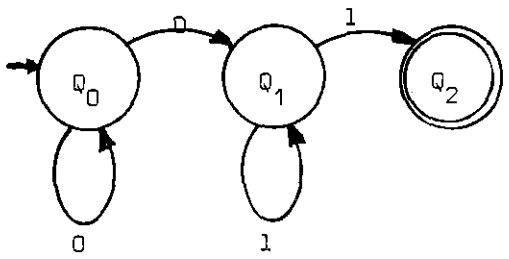
A transition network is a sort of control structure, that is a pattern for a series of actions. By means of a transition network we can 'scan' a word and tell at the end whether it is accepted or not. To do this we proceed as follows.
Take for example the word '110100'. The first state is the start state in this case Q_0. Then we look at the first symbol of the word (here 1) and follow the line in the diagram with that particular symbol as a label. Doing so we arrive in Q_1 : $Q_0 \xrightarrow{1} Q_1$. We repeat this action and obtain the following sequence of transitions: $Q_1 \xrightarrow{1} Q_0 \xrightarrow{0} Q_3 \xrightarrow{1} Q_2 \xrightarrow{0} Q_1 \xrightarrow{0} Q_2$. At this moment there are no symbols left in the word and also we are in the state Q_2. (Note that Q_2 is the final state). If this is the case then we say that the word is accepted by that particular t.n..
As a second example consider the word '1010'. The sequence of states is: $Q_0 \xrightarrow{1} Q_1 \xrightarrow{0} Q_2 \xrightarrow{1} Q_3 \xrightarrow{0} Q_0$. The word is not accepted because Q_0 is not a final state.
The set accepted by the t.n. is the set of all sequences in $\{0,1\}^*$, i.e. combinations with the alphabet 0,1 , containing at least one 0 and one 1 but an odd number of 0's and an odd number of 1's.

Problems 1.1.
(i) Are the words '10100', '010' and '111000' accepted by D ?
(ii) Turn D into a t.n. accepting all sequences in $\{0,1\}^*$ containing both an even number of 0's and an even number of 1's .
(iii) Construct a t.n. accepting all strings in $\{0,1\}^*$ such that there is a 1 immediately following a 0.
(iv) Construct a t.n. accepting all sequences of $a^m b c^n$ and $n,m \geq 0$.

(v) Construct a t.n. accepting all strings in $\{0,1\}^*$ containing any number of 1's and one and only one 0.

Definition 1.2. A simple transition network is called deterministic iff there are no states from which more than one line is leaving with the same label.
A simple transition network is called nondeterministic if it is not deterministic.

All the examples (included the problems) so far discussed are determininistic t.n.'s.
Example 1.2. The following t.n. accepting strings 0^n1^m for $m,n \geq 1$ is a nondeterministic t.n.:



Representation The tabular representation of a t.n. is a list of triples where the first element denotes the label or condition, the second the first state and the third the state after the transition is made.
The order of the list is irrelevant and we number the triples for convenience.

The tabular representation of the t.n. in example 1.1. is:

1. $\langle 1, Q_0 , Q_1 \rangle$
2. $\langle 1, Q_1 , Q_0 \rangle$
3. $\langle 0, Q_1 , Q_2 \rangle$
4. $\langle 0, Q_2 , Q_1 \rangle$
5. $\langle 1, Q_2 , Q_3 \rangle$
6. $\langle 1, Q_3 , Q_2 \rangle$
7. $\langle 0, Q_3 , Q_0 \rangle$
8. $\langle 0, Q_0 , Q_3 \rangle$

We will often leave out brackets and comma's.
The tabular representation of the t.n. in example 1.2. is:

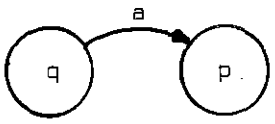
1. $\langle 0, Q_0 , Q_0 \rangle$
2. $\langle 0, Q_0 , Q_1 \rangle$
3. $\langle 1, Q_1 , Q_1 \rangle$
4. $\langle 1, Q_1 , Q_2 \rangle$

A triple in the set will be called a rule r. The i-th triple is denoted as r_i and the j-th element $r_{i,j}$. E.g. in the last tabular representation $r_{1,2} = Q_0$, $r_{3,3} = Q_1$ and $r_{4,1} = 1$.

2. FINITE AUTOMATA

An equivalent way of stating the same information as was expressed in a t.n. is the following.

Definition 1.3. Let M be a system called a finite automaton or finite state machine (for short f.a.), specified by a quintuple $\langle K,\Sigma,\delta,Q_0,E\rangle$. Where K is a finite nonempty set of states. Σ is a finite alphabet, δ is a mapping from K X Σ into K, Q_0 in K is the initial state and $E \subseteq K$ is the set of final states.
δ is called a transition function. It has the form $\delta(q,a) = p$ and q is the first state, p the second and $a \in \Sigma$ is the 'condition' or input symbol. The expression $\delta(q,a) = p$ corresponds to the t.n.:



Note that δ is a recursive function, namely $\delta(q,\lambda) = q$ (λ is the empty string) and $\delta(q,xa) = \delta(\delta(q,x),a)$ where $x \in \Sigma^*$ en $a \in \Sigma$.
If for a word the final state reached after execution of $\delta(q,x)$ is in E, x is accepted.

More formal. Let a configuration be a pair $\langle \alpha,\beta\rangle$ and α is a substring of the input and β the state active when reading this input. We define the relation $\vdash$ between two configurations such that if $\langle ax,\beta\rangle$ is a configuration then $\langle ax,\beta\rangle \vdash \langle x,\gamma\rangle$ if there is a $\delta(a,\beta) = \gamma$.
$\vdash^*$ is the reflexive transitive closure of $\vdash$. The language accepted by an automaton A is $L(A) = \{x \mid \langle x,Q_0\rangle \vdash^* \langle \lambda,\gamma\rangle \quad , \gamma \in E\}$

Example 1.3. The particular instance of a f.a. equivalent with the t.n. of example 1.1. is $M = \langle K, \Sigma, \delta, Q_0, E\rangle$ where $K = \{Q_0,Q_1,Q_2,Q_3\}$, $\Sigma = \{0,1\}$, $E = \{Q_2\}$ and

$\delta(Q_0,0) = Q_3$	$\delta(Q_2,0) = Q_1$
$\delta(Q_0,1) = Q_1$	$\delta(Q_2,1) = Q_3$
$\delta(Q_1,0) = Q_2$	$\delta(Q_3,0) = Q_0$
$\delta(Q_1,1) = Q_0$	$\delta(Q_3,1) = Q_2$

Let us do an example with the word '10'. $\delta(Q_0,10) = \delta(\delta(Q_0,1),0) = \delta(Q_1,0) = Q_2$.
'10' is accepted because Q_2 is in E.

As one can see a finite automaton is just an algebraïc definition of a transition network. The reader is advised to rewrite as an exercise every t.n. so far discussed into the corresponding f.a..

Definition 1.4. A finite automaton $M = \langle K, \Sigma, \delta, Q_0, E\rangle$ is deterministic if δ is a mapping of K X Σ into K.
A finite automaton $M = \langle K, \Sigma, \delta, Q_0, E\rangle$ is nondeterministic if δ is a mapping of K X Σ into subsets of K.

So the difference is that $\delta(q,a)$ is in the case of nondeterministic automata resulting in a set of states and not just only in one state.

Example 1.4. The equivalent automaton of example 1.2. is:
$M = (K, \Sigma, \delta, Q_0, E)$ and $K = \{Q_0, Q_1, Q_2\}$, $\Sigma = \{0,1\}$, $E = \{Q_2\}$ and
$\delta(Q_0,0) = \{Q_0, Q_1\}$
$\delta(Q_1,1) = \{Q_1, Q_2\}$

Obviously M is nondeterministic.

3. REGULAR GRAMMARS

Another related system which is equivalent with a t.n. (and thus with a f.a.) is a regular grammar.

Definition 1.5. Let $G = (Vn, Vt, P, S)$ be a system called a regular grammar, where Vn is a finite nonempty set of symbols called the nonterminal symbols, Vt is a finite nonempty set of symbols called the terminal symbols and $Vn \cap Vt = \emptyset$. P is a set of productions of the form $A \to a\,B$ or $A \to a$ where $A,B \in Vn$ and $a \in Vt$. S is one distinct symbol from Vn called the start symbol or initial axiom. The symbol '$\to$' means 'is rewritten as' and in any string where a symbol appearing on the left hand side is present, we can replace (rewrite) this symbol by the right hand side of that production.
Starting from the axiom S and rewriting until no elements of Vn are left, we can generate a string of the language.

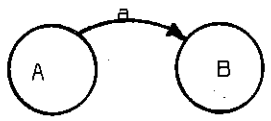
More formal, if $v = xAy$ and there is a production $A \to \alpha$ then $xAy \Rightarrow x\alpha y$.
$\overset{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$ and L(G) (the language described by G) is $\{x \mid S \overset{*}{\Rightarrow} x \text{ and } x \in Vt\}$.

Example 1.5. The particular instance of a regular grammar equivalent with the t.n. of problem 1.(ii) is $G = (Vn, Vt, P, S)$ where $Vn = \{Q_0, Q_1, Q_2, Q_3\}$ $Vt = \{0,1\}$ $S = Q_0$ and P:

1. $Q_0 \to 0\,Q_3$
2. $Q_1 \to 0\,Q_2$
3. $Q_2 \to 0\,Q_1$
4. $Q_3 \to 0$
5. $Q_3 \to 0\,Q_0$
6. $Q_0 \to 1\,Q_1$
7. $Q_1 \to 1\,Q_0$
8. $Q_1 \to 1$
9. $Q_2 \to 1\,Q_3$
10. $Q_3 \to 1\,Q_2$

We generate the word '11010010' as follows. The label on $\Rightarrow$ is the number of the rule applied.
$Q_0 \overset{6}{\Rightarrow} 1Q_1 \overset{7}{\Rightarrow} 11Q_0 \overset{1}{\Rightarrow} 110Q_3 \overset{10}{\Rightarrow} 1101Q_2 \overset{3}{\Rightarrow} 11010Q_1 \overset{2}{\Rightarrow} 110100Q_2 \overset{9}{\Rightarrow} 1101001Q_3 \overset{4}{\Rightarrow} 11010010$.
The reader can verify that the word '101' can not be generated by this grammar.

A production A → a B is equivalent with a transition function $\delta(A,a) = B$ or with the t.n. :



If B is a final state then besides A → aB also the production A → a must be present.

By this method we can easily rewrite regular grammars into transition networks or finite automata or vice-versa. The reader is advised to rewrite all t.n.'s so far discussed into regular grammars as an exercise.

Definition 1.6. A regular grammar is called deterministic if there are no productions with the same nonterminal at the left AND with the same terminal as first symbol on the right.
A regular grammar is called nondeterministic if it is not deterministic.

All grammars so far discussed are nondeterminstic

Example 1.6. Let $G = (Vn, Vt, P, S)$ be a regular grammar and $Vn = \{Q_0, Q_1, Q_2, Q_5, Q_3, Q_4\}$ $Vt = \{0,1\}$, $S = Q_0$ and P:

1. $Q_0 \rightarrow 0\ Q_1$
2. $Q_1 \rightarrow 0\ Q_3$
3. $Q_2 \rightarrow 0\ Q_4$
4. $Q_3 \rightarrow 0\ Q_1$
5. $Q_4 \rightarrow 0$
6. $Q_0 \rightarrow 1\ Q_1$
7. $Q_1 \rightarrow 1\ Q_5$
8. $Q_2 \rightarrow 1\ Q_3$
9. $Q_3 \rightarrow 1\ Q_2$
10. $Q_5 \rightarrow 1$

Obivously G is deterministic.

4. COMMENT

A lot is known about the previously described systems. We have characterization theorems, we know that the languages accepted or generated by these systems form a Boolean algebra of sets, we have sound mathematical proofs about their equivalence and so on. The reader interested in these matters is referred to Hopcroft & Ullman (1969), Minski (1967), Salomaa (1973).

In this paper we will further concentrate on one topic: how can these systems be set to work. For this purpose we will introduce and investigate formal systems called parsers, which act as meta-systems. Basic emphasis will be laid on 'recognition' rather than 'generation'.

5. PARSERS

Although the systems in the previous paragraphs were clearly defined, the actions undertaken for recognizing strings of a language by means of these systems were rather vaguely defined.

To overcome this situation we now introduce a device called a parser. A parser is a meta-system , it takes as data a transition network, a finite state machine, or a regular grammar and then performs certain jobs on a given input such as decide whether given strings are in the language, or extract structural information from the parsing proces, and so on.

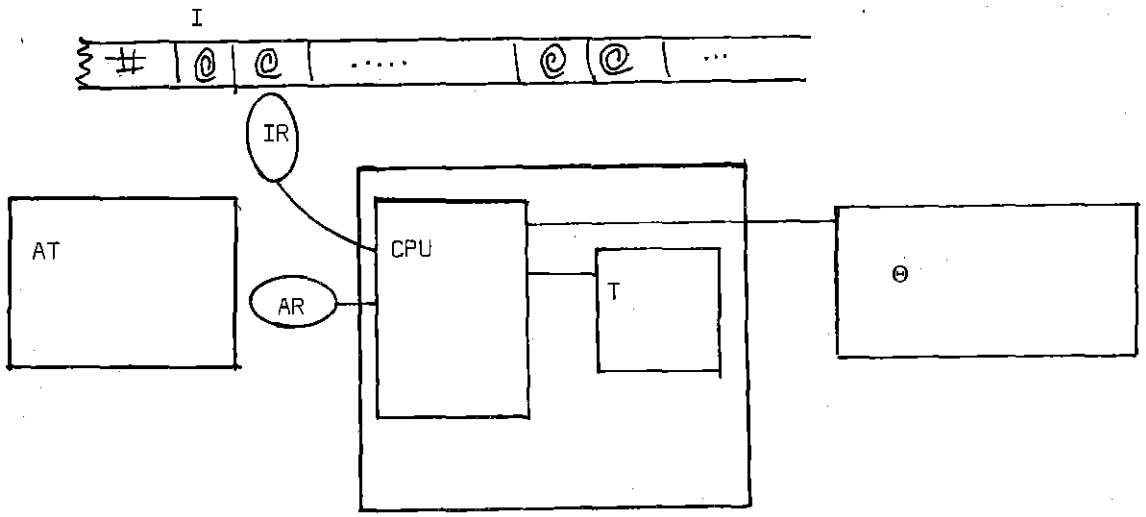
A parser consists of 4 main parts:
(i) A generating or recognizing system such as a t.n., a f.a., or a regular grammar, called the driver of the parser.
(ii) A set of tasks T , these tasks are n-tuples containing all sorts of information. Tasks are created by the execution of previous tasks. In the definition of a parser, one defines the form these tasks will taken.
(iii) A function N in which it is stated how the execution of tasks must proceed. N has 3 parts: (a) the base where t_1, the first task, is given by definition, (b) the recursion or recursive step , where it is defined how a task can be computed from another one, and (c) the restriction where a certain condition is stated for (b) to take place.
(iv) Finally a parser has a means to filter out the required information from the set of tasks.

In summary:
Definition 1.17. A parser $\mathcal{J}$ is defined by a quadruple $\langle A, t_i, N, F \rangle$ where
(i) A is either a t.n., or a f.a. or a regular grammar. A is called the control structure or driver.
(ii) t_i is an n-tuple called a task
(iii) N is a function computing new tasks from old ones
(iv) F is a function $F: T \rightarrow \Theta$ where T is the set of tasks and Θ is a set of structural descriptions. F is called the filter.

To make the story complete, we need a final meta-system (in fact a meta-meta-system seen from the control structure) on which parsers can be executed, this system will be called the parsing machine.
Definition 1.8. A parsing machine PM is defined by a 7-tuple $\langle I, IR, AT, AR, CPU, T, \Theta \rangle$ and
(i) I is a linear input tape containing symbols of the alphabet
(ii) IR is a device reading symbols from I
(iii) AT is a place to store the control structure
(iv) AR is a reading device for AT
(v) CPU is the central processing unit

(vi) T is a set of tasks produced during the computation
(vii) Θ is the resulting output.

Graphically:



In general a parser is performed on a parsing machine as follows:

(i) Initially:
- the underlying control structure A is stored in AT
- the central processing unit is programmed to perform the functions specified in N and F.
- an inputword is written on I
- the initial task t_1 is stored in T

(ii) Second: The parsing machine is set to work such that the central processing unit creates new tasks by executing tasks from T according to the functions in N. These new tasks are again being stored on T and so on. The computation involves reading of the inputtape I and consultation of the control structure A.

(iii) Finally, when no tasks on T are left to be carried out, the central processing unit computes Θ by way of F.

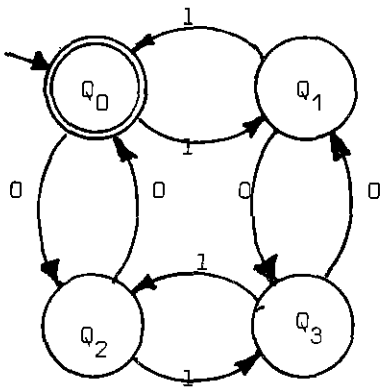
We are not interested here in the formal properties or the power of the parsing machine, it is clear however that it is some sort of register machine. In practice we simulate parsing machines on the currently available computers.

For the rest of the text we will assume implicitly the parsing machine.

6. TYPE 3 PARSERS

Definition 1.9. A parser $\mathcal{S} = (A, t_i, N, F)$ is called a transition network parser, a f.a. parser or a regular grammar parser if A is resp. a transition network, a f.a. or a regular grammar.

Example 1.7. Let $\mathcal{S} = \langle A, t_i, N, F \rangle$ be a t.n. parser such that
(i) A is the following t.n. (cfr. problem 1.(ii))



In tabular form:

1. $\langle 0, Q_0, Q_2 \rangle$
2. $\langle 0, Q_1, Q_3 \rangle$
3. $\langle 0, Q_2, Q_0 \rangle$
4. $\langle 0, Q_3, Q_1 \rangle$
5. $\langle 1, Q_0, Q_1 \rangle$
6. $\langle 1, Q_1, Q_0 \rangle$
7. $\langle 1, Q_2, Q_3 \rangle$
8. $\langle 1, Q_3, Q_2 \rangle$

(ii) t_i is an ordered pair $\langle a_{i,1}, a_{i,2} \rangle$
where $a_{i,1}$ is the position on I to be read by IR
$a_{i,2}$ is the state active when executing the task

(iii) N is defined as follows:

1. The base $t_1 = \langle 1, Q_0 \rangle$ (Q_0 is the start state)
2. The recursion step: t_{i+1} is computed from t_i by N and

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{i+1,1} \\ r_{k,3} & \text{for } Y = a_{i+1,2} \end{cases}$$

where $r_{k,1} = I(a_{i,1})$ and $r_{k,2} = a_{i,2}$

3. The restriction:

N is defined iff $a_{i,1} + 1 \leq |\sigma| + 1$ where σ is the inputword and $|\sigma|$ denotes the length of σ, and $\exists k\ (I(a_{i,1}) = r_{k,1}),\quad 8 \geqslant k > 0$

(iv) $F: T \rightarrow \Theta$ where Θ is either 0 or 1. If the last task $t_i = \langle a_{i,1}, a_{i,2} \rangle$ is such that $a_{i,1} = |\sigma| + 1$ and $a_{i,2}$ is a final state in A, then $\Theta = 0$ else $\Theta = 1$.

(0 when accepted, 1 if not accepted).

The recursion step should be understood as follows: N is a function computing n-tuples from n-tuples. t_i is the input and t_{i+1} is the result of the computation. Y is a dummy element for the elements of the n-tuple. Thus the function results in $a_{i,1}+1$ for $Y = a_{i+1,1}$. This means the first element in the n-tuple t_{i+1}, i.e. $a_{i+1,1}$ becomes $a_{i,1} + 1$. The second element of t_{i+1}, namely $a_{i+1,2}$ becomes $r_{k,3}$ with the requirement that $r_{k,3} = I(a_{i,1})$ and $r_{k,2} = a_{i,2}$

Example 1.7.1.

$\sigma = 1001$, $|\sigma| = 4$

$t_1 = \langle 1, Q_0 \rangle$ (the base)

We compute t_2 from t_1 by the recursion step:

$$(N(t_1))(Y) = \begin{cases} a_{1,1} + 1 = 2 & \text{for } Y = a_{1+1,1} = a_{2,1} \\ r_{5,3} = Q_1 & \text{for } Y = a_{1+1,2} = a_{2,2} \\ \quad \text{because } r_{5,1} = I(a_{1,1}) = I(1) = 1 & \\ \quad \text{and } r_{5,2} = a_{1,2} = Q_0 & \end{cases}$$

So: $t_2 = \langle 2, Q_1 \rangle$

$t_3 = \langle 3, Q_3 \rangle$

$t_4 = \langle 4, Q_1 \rangle$

$t_5 = \langle 5, Q_0 \rangle$

t_6 is undefined because $a_{5,1} + 1 > |\sigma| + 1$ (restriction)

Filter:

Θ is 0 because $a_{5,1} = |\sigma| + 1$, $a_{5,2}$ is Q_0 and Q_0 is a final state. Conclusion σ is accepted by the transition network.

Example 1.7.2.

$\sigma = 1000$, $|\sigma| = 4$

$t_1 = \langle 1, Q_0 \rangle$

$t_2 = \langle 2, Q_1 \rangle$

$t_3 = \langle 3, Q_3 \rangle$
$t_4 = \langle 4, Q_4 \rangle$
$t_5 = \langle 5, Q_3 \rangle$

t_6 is undefined and $\Theta = 1$ because Q_3 is not a final state.

Example 1.7.3.

σ = 1010011101 , $|\sigma| = 10$

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_1 \rangle$
$t_3 = \langle 3, Q_3 \rangle$
$t_4 = \langle 4, Q_2 \rangle$
$t_5 = \langle 5, Q_0 \rangle$
$t_6 = \langle 6, Q_2 \rangle$
$t_7 = \langle 7, Q_3 \rangle$
$t_8 = \langle 8, Q_2 \rangle$
$t_9 = \langle 9, Q_3 \rangle$
$t_{10} = \langle 10, Q_1 \rangle$
$t_{11} = \langle 11, Q_0 \rangle$

t_{12} is undefined and $\Theta = 0$ because Q_0 is a final state.

It is easy to see that a parser (at this stage) simply mimics the behavior of the actions described when deciding whether a string is accepted or not by a t.n. (cf. p.3). Note also that the input is in fact independent of the parsing system. We could have taken another t.n. as well. (Of course there must always be one).
An interesting thing is also that we can use the same construct $\mathcal{P}$ not only for a t.n. but also for a f.s. machine and for a regular grammar.

Example 1.8.

Let $\mathcal{P} = \langle A, t_i, N, f \rangle$ be a f.a. parser such that
(i) $A = \langle K, \Sigma, \delta, Q_0, E \rangle$ is a f.a. and
$K = \{Q_0, Q_1, Q_2, Q_3\}$
$\Sigma = \{0,1\}$
$E = \{Q_0\}$
and
$\delta(Q_0,0) = Q_2$ $\quad$ $\delta(Q_0,1) = Q_1$
$\delta(Q_1,0) = Q_3$ $\quad$ $\delta(Q_1,1) = Q_0$
$\delta(Q_2,0) = Q_0$ $\quad$ $\delta(Q_2,1) = Q_3$
$\delta(Q_3,0) = Q_1$ $\quad$ $\delta(Q_3,1) = Q_2$

(ii) t_i is an ordered pair $\langle a_{i,1}, a_{i,2} \rangle$
where $a_{i,1}$ is the position on I to be read by IR
$a_{i,2}$ is the state active when executing the task

(iii) N is defined as follows:
1. the base $t_1 = \langle 1, Q_0 \rangle$ (Q_0 is the start state)
2. the recursion step
t_{i+1} is computed from t_i by N and

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{i,1} \\ \delta(a_{i,2}, I(a_{i,1})) & \text{for } Y = a_{i,2} \end{cases}$$

3. the restriction

N is defined iff $a_{i,1} + 1 \leq |\sigma| + 1$ where σ is the input word and $|\sigma|$ denotes the length of σ and iff $I(a_{i,1}) \in \Sigma$.

(iv) $F:T \rightarrow \Theta$ where Θ is either 0 or 1. If the last task $t_i = \langle a_{i,1}, a_{i,2} \rangle$ is such that $a_{i,1} = |\sigma| + 1$ and $a_{i,2} \notin E$ then $\Theta = 0$ else $\Theta = 1$.

Example 1.8.1.

σ = 1001 , $|\sigma| = 4$

T:

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_1 \rangle$
$t_3 = \langle 3, Q_3 \rangle$
$t_4 = \langle 4, Q_1 \rangle$
$t_5 = \langle 5, Q_0 \rangle$

t_6 is undefined because $a_{6,1} + 1 > |\sigma| + 1$

Θ is 0 because $a_{5,1} = |\sigma| + 1$ and $a_{5,2} \in E$. In other words $\sigma \in L(A)$ and L(A) is the language accepted by the automation.

Note that the only difference between ex. 1.7. and ex. 1.8. the controlstructure is and the way in which the control structure is consulted in N. The reader is advised to try ex. 1.7.2. and ex. 1.7.3. on this parser.

Note also that the set of tasks T produced during the computation is the same as in example 1.7.1..

Example 1.9.

Let $S = \langle A, t_i, N, F \rangle$ be a regular grammar parser and

(i) $A = \langle Vn, Vt, P, S, \rangle$ is a regular grammar where

$Vn = \{Q_0, Q_1, Q_2, Q_3, Q_4, Q_5\}$
$Vt = \{0,1\}$
$S = Q_0$

P:
1. $Q_0 \rightarrow 0\ Q_2$
2. $Q_1 \rightarrow 0\ Q_3$
3. $Q_2 \rightarrow 0\ Q_4$
4. $Q_3 \rightarrow 0\ Q_1$
5. $Q_4 \rightarrow 0$
6. $Q_0 \rightarrow 1\ Q_1$
7. $Q_1 \rightarrow 1\ Q_5$
8. $Q_2 \rightarrow 1\ Q_3$
9. $Q_3 \rightarrow 1\ Q_2$
10. $Q_5 \rightarrow 1$

(ii) t_i is a pair $\langle a_{i,1}, a_{i,2} \rangle$
where $a_{i,1}$ = the position to be read on the input task

$a_{i,2}$ = the nonterminal at this task

(iii) N is defined as follows:

1. the base $t_1 = \langle 1, Q_0 \rangle$
   (Q_0 is the start symbol)
2. the recursion step: t_{i+1} is computed from t_i by N and

$$(N(t_i))(Y) = \begin{cases} a_{i,1}+1 & \text{for } Y = a_{i+1,1} \\ \text{The last symbol of a procuction } \pi \text{ where the left side of } \pi \text{ is } a_{i,2} \text{ and the first symbol of the right is } I(a_{i,1}) & \text{for } Y = a_{i+1,2} \end{cases}$$

3. the restriction:
   N is defined iff $a_{i,1} + 1 \leq |\sigma| + 1$ where σ is the inputword and $|\sigma|$ denotes the length of σ, iff $I(a_{i,1}) \in Vt$ and iff $a_{i,2} \neq \lambda$.

(iv) F: $T \to \Theta$ where Θ is either 0 or 1. If the last task $t_i = \langle a_{i,1}, a_{i,2} \rangle$ is such that $a_{i,1} = |\sigma| + 1$ and $a_{i,2} = \lambda$ then $\Theta = 0$, else $\Theta = 1$. (means cannot be generated).

(Note if a production is of the form $A \to a$ then we assume λ (the empty string) after a: $A \to a\lambda$)

Example 1.9.1.

$\sigma = 000$, $|\sigma| = 3$

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_2 \rangle$
$t_3 = \langle 3, Q_4 \rangle$
$t_4 = \langle 4, \lambda \rangle$

Θ is 0 because t_4 is such that $a_{4,1} = |\sigma| + 1$ and $a_{4,2} = \lambda$ therefore σ is generate by A.

Example 1.9.2.

$\sigma = 01100$, $|\sigma| = 5$

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_2 \rangle$
$t_3 = \langle 3, Q_3 \rangle$
$t_4 = \langle 4, Q_2 \rangle$
$t_5 = \langle 5, Q_4 \rangle$
$t_6 = \langle 6, \lambda \rangle$

Θ is 0 because $t_{6,1} = |\sigma| + 1$ and $t_{6,2} = \lambda$

Example 1.9.3.

$\sigma = 1110$ $|\sigma| = 4$

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_1 \rangle$
$t_3 = \langle 3, Q_5 \rangle$
$t_4 = \langle 4, \lambda \rangle$

t_5 is undefined because $t_{4,2} = \lambda \cdot \Theta$ is 1 because in the last task $a_{4,1} \neq |\sigma| + 1$.

Again the similarity between example 1.9. and example 1.7. and 1.8. should be obvious. For this reason we will call t.n. parsers, f.a. parsers and regular grammar parsers type 3 parsers. Moreover in the example we always used a deterministic control structure . Therefore the parsers of example 1.7., 1.8. and 1.9. belong to the class of deterministic type 3 parsers.

Definition 1.10. A parser $\mathcal{S} = (A, t_i, N, F)$ is a deterministic type 3 parser if A is a deterministic t.n. or a deterministic f.a. or a deterministic regular grammar.

Before we discuss in further detail what we can do with parsers besides deciding whether a string is in the language or not, we will construct nondeterministic type 3 parsers.

7. NONDETERMINISTIC TYPE 3 PARSERS

Definition 1.11. A parser $\mathcal{S} = (A, t_i, N, F)$ is a nondeterministic type 3 parser iff A is a nondeterministic t.n., a nondeterministic finite state machine or a nondeterministic regular grammar.

The only difference between deterministic and nondeterministic type 3 parsers is that instead of executing function N only once, it must be executed as many times as this is possible. So a nondeterministic t.n. parser is equivalent with a deterministic t.n. parser except for the recursion step where N must be executed for every rule r where $r_{k,1} = I(a_{i,1})$ and $r_{k,2} = a_{i,2}$. Similarly for a nondeterministic f.a. parser N must be executed for every element in the set resulting from $\delta(a_{i,2}, I(a_{i,1}))$ and for a regular grammar parser N must be executed for every production π where the left side is equal to $a_{i,2}$ and the first symbol on the right side is equal to $I(a_{i,1})$.
From now on v will be a variable denoting the number of tasks created; each time a new task is created v is augmented by 1. Also instead of the last task we will require just the presence of the task specified in F.

We give in full detail a nondeterministic f.a. parser and expect from the reader that he will construct a nondeterministic t.n. and a nondeterministic regular grammar parser.

Example 1.10. Let $\mathcal{S} = (A, t_i, N, F)$ be a nondeterministic f.a. parser and
(i) $A = (K, \Sigma, \delta, Q_0, E)$ is a nondeterministic f.a. where $K = \{Q_0, Q_1, Q_2, Q_3, Q_4\}$
$\Sigma = \{0,1\}$, $E = \{Q_2, Q_4\}$ and

$\delta(Q_0,0) = \{Q_0, Q_3\}$ $\quad$ $\delta(Q_0,1) = \{Q_0, Q_1\}$
$\delta(Q_1,0) = \emptyset$ (the empty set)
$\delta(Q_2,0) = \{Q_2\}$ $\quad$ $\delta(Q_1,1) = \{Q_2\}$
$\delta(Q_3,0) = \{Q_4\}$ $\quad$ $\delta(Q_2,1) = \{Q_2\}$

$\delta(Q_4,0) = \{Q_4\}$ $\qquad$ $\delta(Q_3,1) = \emptyset$

$\qquad\qquad\qquad\qquad\quad \delta(Q_4,1) = \{Q_4\}$

(ii) t_i is a pair $\langle a_{i,1}, a_{i,2} \rangle$

where $a_{i,1}$ = the position to be read on the input tape

$a_{i,2}$ = the state of the automaton at this task

(iii) N is defined as follows:

1. the base : $t_1 = \langle 1, Q_0 \rangle$ and $v = 1$

2. the recursion:

For every element in the set resulting from $\delta(a_{i,2}, I(a_{i,1}))$ a new task ($v = v + 1$) is made (if the set is empty then no tasks are made), such that:

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ \delta(a_{i,2}, I(a_{i,1})) & \text{for } Y = a_{v,2} \end{cases}$$

3. the restriction:

N is defined iff $a_{i,1} + 1 \leqslant |\sigma| + 1$ where σ is the inputword and $|\sigma|$ denotes the length of σ and iff $I(a_{i,1}) \in \Sigma$

(iv) Finally:

$F: T \to \Theta$ where $\Theta \in \{0,1\}$ and $\Theta = 0$ iff there is a task in T, t_i, such that $a_{i,1} = |\sigma| + 1$ and $a_{i,2} \in E$.

Example 1.10.1.

$\sigma = 1100$, $|\sigma| = 4$

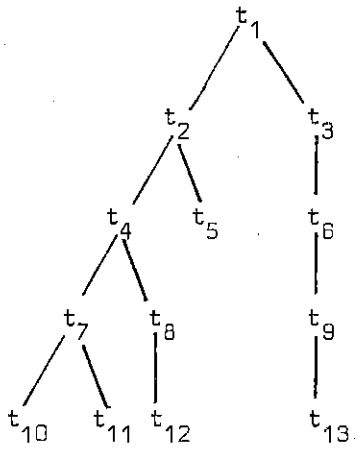
$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_0 \rangle$
$t_3 = \langle 2, Q_1 \rangle$
$t_4 = \langle 3, Q_0 \rangle$
$t_5 = \langle 3, Q_1 \rangle$
$t_6 = \langle 3, Q_2 \rangle$
$t_7 = \langle 4, Q_0 \rangle$
$t_8 = \langle 4, Q_3 \rangle$
$t_9 = \langle 4, Q_2 \rangle$
$t_{10} = \langle 5, Q_0 \rangle$
$t_{11} = \langle 5, Q_3 \rangle$
$t_{12} = \langle 5, Q_4 \rangle$
$t_{13} = \langle 5, Q_2 \rangle$

$\sigma \in L(A)$ because t_{12} is a task having in $a_{i,1}$ $|\sigma| + 1$ and in $a_{i,2} \in E$. Also t_{13} has this property . If this is the case then we say that the word is ambiguous according to A.

Definition 1.12. The structural description of the parsing process will be denoted as a labeled plane-rooted tree where the top node is the first task (t_1) and for all tasks emerging by N from a task t_i , we draw lines between these tasks and t_i.

Example 1.11.

The s.d. for the parsing process performed in example 1.10.1. is:

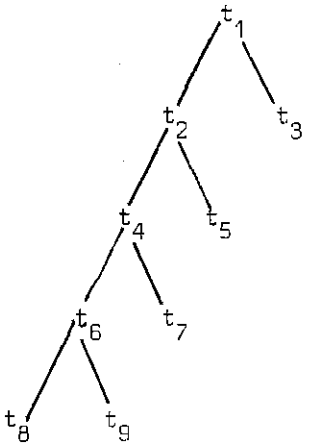


Example 1.10.2.

σ = 0101 , $|\sigma|$ = 4

$t_1 = (1, Q_0)$
$t_2 = (2, Q_0)$
$t_3 = (2, Q_3)$
$t_4 = (3, Q_0)$
$t_5 = (3, Q_1)$
$t_6 = (4, Q_0)$
$t_7 = (4, Q_3)$
$t_8 = (5, Q_0)$
$t_9 = (5, Q_1)$

the s.d. for this parsing process:



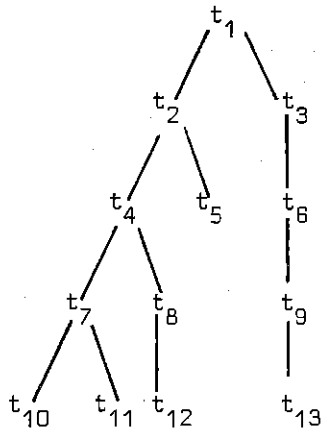
σ is not in L(A) because there is no task t_i where $a_{i,2} \in E$ and $a_{i,1} = |\sigma| + 1$

Example 1.10.3.

σ = 0011

$t_1 = (1, Q_0)$
$t_2 = (2, Q_0)$
$t_3 = (2, Q_3)$

$t_4 = \langle 3, Q_0 \rangle$
$t_5 = \langle 3, Q_3 \rangle$
$t_6 = \langle 3, Q_4 \rangle$
$t_7 = \langle 4, Q_0 \rangle$
$t_8 = \langle 4, Q_1 \rangle$
$t_9 = \langle 4, Q_4 \rangle$
$t_{10} = \langle 5, Q_0 \rangle$
$t_{11} = \langle 5, Q_1 \rangle$
$t_{12} = \langle 5, Q_2 \rangle$
$t_{13} = \langle 5, Q_4 \rangle$



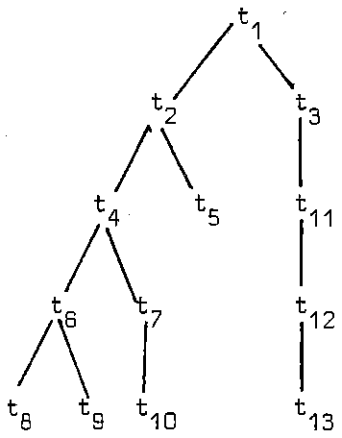
t_{12} and t_{13} are both tasks where $a_{i,1}$ is $|\sigma| + 1$ and $a_{i,2} \in E$ therefore σ is in L(A).

Note that the order in which the tasks are carried out is of no importance (but they should all be carried out once !). To illustrate this we do an alternative way for example 1.10.3..

Example 1.10.4.

σ = 0011

$t_1 = \langle 1, Q_0 \rangle$
$t_2 = \langle 2, Q_0 \rangle$
$t_3 = \langle 2, Q_3 \rangle$
$t_4 = \langle 3, Q_0 \rangle$
$t_5 = \langle 3, Q_3 \rangle$
$t_6 = \langle 4, Q_0 \rangle$
$t_7 = \langle 4, Q_1 \rangle$
$t_8 = \langle 5, Q_0 \rangle$
$t_9 = \langle 5, Q_1 \rangle$
$t_{10} = \langle 5, Q_2 \rangle$
$t_{11} = \langle 3, Q_4 \rangle$
$t_{12} = \langle 4, Q_4 \rangle$
$t_{13} = \langle 5, Q_4 \rangle$



In this case t_{10} and t_{13} are the tasks fulfilling the conditions in F, therefore the result is the same. Note that the structure of the parsing process is the same, only the indices of the tasks are different.

8. SOME APPLICATIONS OF THE FILTER

Now we try to show that we can do better with parsers than just say that something is in the language or not. We want to remember after the computation HOW its was done.

For example we want to find out whether the input string is ambiguous or what the possible ways of deriving a string are, what the structural description is, what rules were used to generate a given string , in what states the automaton was, etc... We will give some examples but expect the reader to try out other applications.

The basic strategy is the following: put in a task all information that is necessary for a given job and then design your filter F such that this necessary information is extracted from the task and then used to obtain the desired result.

Example 1.12.

The aim of this parser is to accept as data a f.a. and to compute the distinct states of the automaton when accepting a given inputstring.

Let $\mathcal{P} = \langle A, t_i, N, F \rangle$ be a nondeterministic f.a. parser where

(i) A is $\langle K, \Sigma, \delta, Q, E \rangle$ (cf. e.g. 1.10.)

$K = \{Q_0, Q_1, Q_2, Q_3, Q_4\}$, $\Sigma = \{0,1\}$, $E = \{Q_2, Q_4\}$ and

$\delta(Q_0,0) = \{Q_0, Q_3\}$ $\quad$ $\delta(Q_0, 1) = \{Q_0, Q_1\}$

$\delta(Q_1,0) = \emptyset$ $\quad$ $\delta(Q_1,1) = \{Q_2\}$

$\delta(Q_2,0) = \{Q_2\}$ $\quad$ $\delta(Q_2,1) = \{Q_2\}$

$\delta(Q_3,0) = \{Q_4\}$ $\quad$ $\delta(Q_3,1) = \emptyset$

$\delta(Q_4,0) = \{Q_4\}$ $\quad$ $\delta(Q_4,1) = \{Q_4\}$

(ii) t_i is a triple $\langle a_{i,1}, a_{i,2}, a_{i,3} \rangle$

where $a_{i,1}$ is the position to be read on the input tape

$a_{i,2}$ is the state of the automaton

$a_{i,3}$ is the index j of the task t_j that was the basis for this task, in other words t_j was the input for N and t_i is the output, $a_{i,3}$ is called the anchor ($\mathfrak{t}$) of t_i.

(iii) N is defined as follows:

1. the base: $t_1 = \langle 1, Q_0, 0 \rangle$ $\quad$ (v = 1)
2. the recursive step:

For every element in the set resulting from $\delta(a_{i,3}, I(a_{i,1}))$ a new task is made (if the set is empty, then no tasks are made), such that(v = v + 1)

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ \delta(a_{i,3}, I(a_{i,2})) & \text{for } Y = a_{v,2} \\ i & \text{for } Y = a_{v,3} \end{cases}$$

3. The restriction: N is defined iff $a_{i,1} \leq |\sigma| + 1$, and iff $I(a_{i,1}) \in \Sigma$

(iv) F is a bit more comples:

We define a valid start task in $\mathcal{S}$ as a task t_i such that $a_{i,1} = 1, a_{i,2} = Q_0$ and $a_{i,3} = 0$.

We define a valid end task in $\mathcal{S}$ as a task t_i such that $a_{i,1} = |\sigma| + 1$ and $a_{i,2} \in E$.

We define a valid path P through T (the set of tasks) as a sequence of indices where

(i) i_1 is the index of a valid end task in T

(ii) $i_{j+1} = a_{i_j,3}$

(iii) if $(a_{i_j,3}) = 0$ then P is complete.

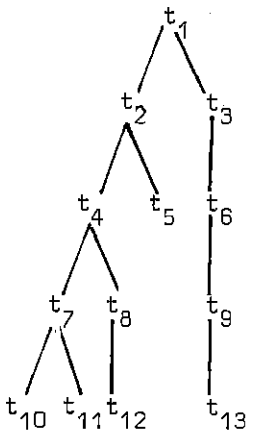
(We give an example of the computation of a valid path in the next example)

Θ is a set of sequences of states $\{a_{p_j,2}\}$ where $p_j \in P$ and j is an index ranging from the last element of a path P to the first one.

Example 1.12.1.

σ = 1100

$t_1 = \langle 1, Q_0, 0 \rangle$
$t_2 = \langle 2, Q_0, 1 \rangle$
$t_3 = \langle 2, Q_1, 1 \rangle$
$t_4 = \langle 3, Q_0, 2 \rangle$
$t_5 = \langle 3, Q_1, 2 \rangle$
$t_6 = \langle 3, Q_2, 3 \rangle$
$t_7 = \langle 4, Q_0, 4 \rangle$
$t_8 = \langle 4, Q_3, 4 \rangle$
$t_9 = \langle 4, Q_2, 6 \rangle$
$t_{10} = \langle 5, Q_0, 7 \rangle$
$t_{11} = \langle 5, Q_3, 7 \rangle$
$t_{12} = \langle 5, Q_4, 8 \rangle$
$t_{13} = \langle 5, Q_2, 9 \rangle$



There are two valid tasks (t_{12} and t_{13}), therefore we have two paths:

$P_{t_{12}} = \langle 12, 8, 4, 2, 1 \rangle$ and $P_{t_{13}} = \langle 13, 9, 6, 3, 1 \rangle$

We give now an explicit example of the computation of such a path, i_1 is the index fof a valid end task in T, here t_{12} is one. So $i_1 = 12$. j is a variable in the recursive step, in the beginning : j = 1.

- $i_{j+1} = a_{i_j,3}$ or $i_{1+1} = a_{i_1,3}$ or $i_2 = a_{12,3}$ (from i = 12)
  according to T: $a_{12,3} = 8$ therefore $i_2 = 8$
- j becomes 2
  $i_{2+1} = a_{i_2,3}$ or $i_3 = a_{8,3} = 4$ Path up to now: $\langle 12,8,4 \rangle$
- j becomes 3
  $i_{3+1} = a_{i_3,3}$ or $i_4 = a_{4,3} = 2$
- j becomes 4
  $i_{4+1} = a_{i_4,3}$ or $i_5 = a_{2,3} = 1$

= j becomes 5:

$i_j,3 = a_{i_5,3} = a_{1,3} = 0$, therefore the path is complete.

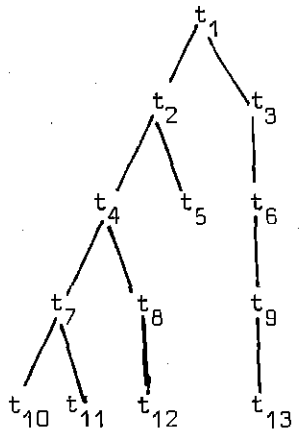
Consequently:

$\Theta = \{\langle Q_0, Q_0, Q_0, Q_3, Q_4 \rangle, \langle Q_0, Q_1, Q_2, Q_2, Q_2 \rangle\}$

Example 1.12.2

σ = 0011

$t_1 = \langle 1, Q_0, 0 \rangle$
$t_2 = \langle 2, Q_0, 1 \rangle$
$t_3 = \langle 2, Q_3, 1 \rangle$
$t_4 = \langle 3, Q_0, 2 \rangle$
$t_5 = \langle 3, Q_3, 2 \rangle$
$t_6 = \langle 3, Q_4, 3 \rangle$
$t_7 = \langle 4, Q_0, 4 \rangle$
$t_8 = \langle 4, Q_1, 4 \rangle$
$t_9 = \langle 4, Q_4, 6 \rangle$
$t_{10} = \langle 5, Q_0, 7 \rangle$
$t_{11} = \langle 5, Q_1, 7 \rangle$
$t_{12} = \langle 5, Q_2, 8 \rangle$
$t_{13} = \langle 5, Q_4, 9 \rangle$



In this case there are again two valid end tasks: t_{12} and t_{13} . The corresponding paths are $P_{t_{12}} = \langle 12, 8, 4, 2, 1 \rangle$ and $P_{t_{13}} = \langle 13,9,6,3,1 \rangle$

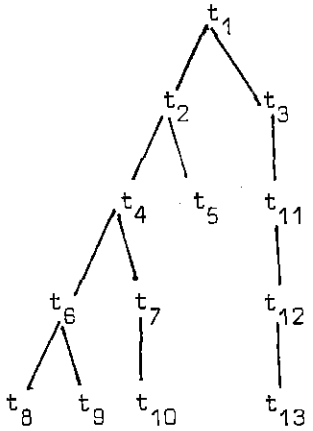
$\Theta = \{\langle Q_0, Q_0, Q_0, Q_1, Q_2 \rangle, \langle Q_0, Q_3, Q_4, Q_4, Q_4 \rangle\}$

Note that the order in which the tasks are carried out is again of no importance. To illustrate this we do an alternative way for example 4.2..

Example 1.12.3.

σ = 0011

$t_1 = \langle 1, Q_0, 0 \rangle$
$t_2 = \langle 2, Q_0, 1 \rangle$
$t_3 = \langle 2, Q_3, 1 \rangle$
$t_4 = \langle 3, Q_0, 2 \rangle$
$t_5 = \langle 3, Q_3, 2 \rangle$
$t_6 = \langle 4, Q_0, 4 \rangle$



$t_7 = \langle 4, Q_1, 4 \rangle$
$t_8 = \langle 5, Q_0, 6 \rangle$
$t_9 = \langle 5, Q_1, 6 \rangle$
$t_{10} = \langle 5, Q_2, 7 \rangle$
$t_{11} = \langle 3, Q_4, 3 \rangle$
$t_{12} = \langle 4, Q_4, 11 \rangle$
$t_{13} = \langle 5, Q_4, 12 \rangle$

The two valid end tasks are t_{10} and t_{13}.

$P_{t_{10}} = \langle 10, 7, 4, 2, 1 \rangle$ and $P_{t_{13}} = \langle 13, 12, 11, 3, 1 \rangle$

$\Theta^{10} = \{\langle Q_0, Q_0, Q_0, Q_1, Q_2 \rangle, \langle Q_0, Q_3, Q_4, Q_4, Q_4 \rangle\}$

As the reader can see the same result is obtained.

Example 1.13.

Now we construct a parser which accepts a regular grammar as data and computes the rules which are necessary for generating a given inputstring. (These data can be important e.g. in developing a probabilistic grammar).

Let $\mathcal{S} = \langle A, t_i, N, P \rangle$ be a nondeterministic regular grammar parser where

(i) A is a regular grammar $\langle Vn, Vt, P, S \rangle$ and

$Vn = \{Q_0, Q_1, Q_2, Q_3\}$, $Vt = \{0,1\}$, $S = Q_0$ and P:

1. $Q_0 \rightarrow 0\ Q_3$	6. $Q_0 \rightarrow 1\ Q_1$
2. $Q_1 \rightarrow 0\ Q_2$	7. $Q_1 \rightarrow 1\ Q_0$
3. $Q_2 \rightarrow 0\ Q_1$	8. $Q_1 \rightarrow 0$
4. $Q_2 \rightarrow 1$	9. $Q_2 \rightarrow 1\ Q_3$
5. $Q_3 \rightarrow 0\ Q_0$	10. $Q_3 \rightarrow 1\ Q_2$

The integers before a production will be denoted by the symbol π.

(ii) t_i is a quadruple $\langle a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4} \rangle$

where $a_{i,1}$ is the position to be read on the input tape
$a_{i,2}$ is the symbol of interest at this task
$a_{i,3}$ is the anchor of the task
$a_{i,4}$ is the rule applied to construct this task

(iii) N is defined as follows:

1. the base: $t_1 = \langle 1, Q_0, 0, 0 \rangle$ $(v = 1)$
2. the recursive step

For every production π where the left side is equal to $a_{i,2}$ and the first symbol on the right side is equal to $I(a_{i,1},1)$; we make a new task $(v = v + 1)$ such that:

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ \text{the last symbol of a production } \pi \text{ where the left side of } \pi \text{ is } a_{i,2} \text{ and the first symbol of the right is } I(a_{i,1}) & \text{for } Y = a_{v,2} \\ i & \text{for } Y = a_{v,3} \\ \pi & \text{for } Y = a_{v,4} \end{cases}$$

3. restriction:

N is defined iff $a_{i,1} + 1 \leq |\sigma| + 1$, $I(a_{i,1}) \in Vt$, $a_{i,2} \neq \lambda$

(iv) F:

A valid start task in $\mathcal{T}$ is a task t_i where $a_{i,1} = 1$ and $a_{i,2} = 0$.

A valid end task in $\mathcal{T}$ is a task t_i where $a_{i,1} = |\sigma| + 1$ and $a_{i,2} = \lambda$

A valid path P through T is a sequence of indices and

(i) i_1 is the index of a valid end task in T

(ii) $i_{j+1} = a_{i_j,3}$

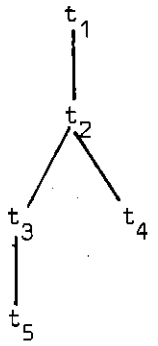
(iii) if $(a_{i_j,3}) = 0$ then P is complete.

Θ is a sequence of indices of productions $:\{(a_{p_j,3})\}$ where $p \in P$ and j is an index ranging from the second last element of a path P to the first one.

Example 1.13.1.

$\sigma = 100$

$t = \langle 1, Q_0, 0, 0 \rangle$
$t = \langle 2, Q_1, 1, 6 \rangle$
$t = \langle 3, \lambda, 2, 8 \rangle$
$t = \langle 3, Q_2, 2, 2 \rangle$
$t = \langle 4, Q_1, 4, 3 \rangle$



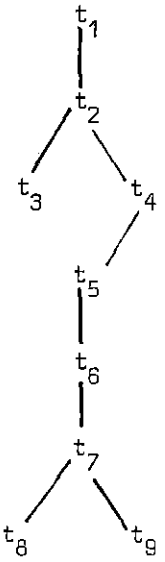
The word is not accepted (cannot be generated by A) because there is no valid end task in T.

Example 1.13.2.

$\sigma = 010101$ $\quad |\sigma| = 6$

$t_1 = \langle 1, Q_1, 0, 0 \rangle$
$t_2 = \langle 2, Q_3, 1, 1 \rangle$
$t_3 = \langle 3, \lambda, 2, 4 \rangle$
$t_4 = \langle 3, Q_2, 2, 10 \rangle$

$t_5 = (4, Q_1, 4, 3)$
$t_6 = (5, Q_0, 5, 7)$
$t_7 = (6, Q_3, 6, 1)$
$t_8 = (7, \lambda, 7, 4)$
$t_9 = (7, Q_2, 7, 9)$



t_8 is a valid end task. A valid path from t_8 is (8, 7, 6, 5, 4, 2, 1)

Finally Θ is the following sequence: (1, 10, 3, 7, 1, 4)

As the reader can see the application of these rules in the generation process indeed results in '010101':

$Q_0 \overset{1}{\Rightarrow} 0\ Q_3 \overset{10}{\Rightarrow} 0\ 1\ Q_2 \overset{3}{\Rightarrow} 0\ 1\ 0\ Q_1 \overset{7}{\Rightarrow} 0\ 1\ 0\ 1\ Q_0 \overset{1}{\Rightarrow} 0\ 1\ 0\ 1\ 0\ Q_3 \overset{4}{\Rightarrow} 0\ 1\ 0\ 1\ 0\ 1$

Problems 1.2.

(i) Construct a grammar generating the set of sequences $\{a,b,c\}^*$ where before and after each b (if there is a b in the string) there is an a.
Then construct a parser where the output is the set of pairs representing a derivation where the first pair contains the start symbol and λ and the last pair the last terminal symbol of the derivation.
Parse c, bac, cabacccabaa.

(ii) Construct an equivalent transition network parser doing the same job.

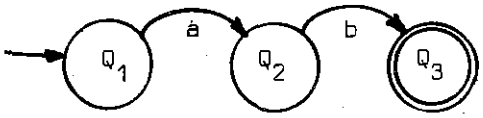
(iii) Construct a regular grammar parser which for an arbitrary regular grammar will decide whether a string is ambiguous according to that grammar or not.

2. Recursive transition networks and related systems

1. RECURSIVE TRANSITION NETWORKS

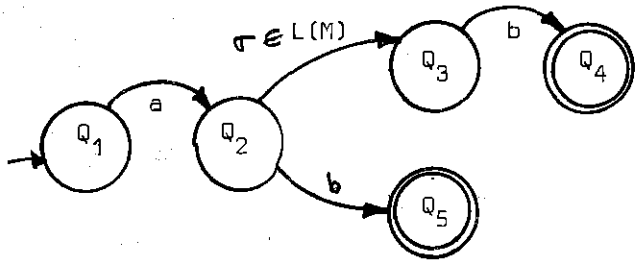
We now introduce a higher class of control structures and consequently a more complex parser.

Consider the following transition network M'.



The language accepted by M': L(M') = ab

Suppose we extend M' into M in the following way:



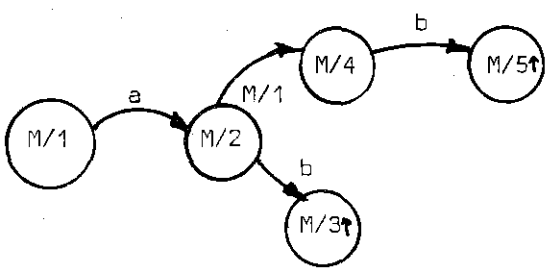
where the condition or label on the directed line going from Q_2 to Q_3 is a word accepted by the transition network M. In other words, through a sort of subroutine call, the network is started again, and if a word is found, the transition can be made.

The language accepted by M, $L(M) = a^n b^n$ for $n \geq 1$.

Def. 2.1. Transition networks where the condition for a certain transition is itself a transition network are called recursive transition networks.

For convenience we denote from now on the states of a network by A/i where A is the name of the transition network and i is the number of the state. For the start state i is always equal to 1 and for any final state we write after the index i an arrow (↑) or just the arrow without an index. Final states are sometimes called pop up states.

M is then written as follows:

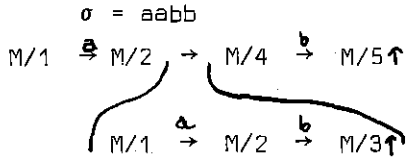


M in tabular representation

1. ⟨a , M/1 , M/2 ⟩
2. ⟨M/1, M/2, M/4 ⟩
3. ⟨b, M/4 , M/5↑ ⟩
4. ⟨b, M/2 , M/3↑ ⟩

From this representation it becomes clear that the symbol on the recursive transition is in fact the first state of another network. Let us do an example of a scanning controled by M

σ = aabb



2. CONTEXT - FREE GRAMMAR

An equivalent way of stating the same information as for a recursive transition network is a context-free grammar (for short cfg.)

Def. 2.2. Let G = ⟨Vn, Vt, P, S⟩ be called a cfg. where Vn is a finite set of non-terminals, Vt is a finite set of terminals and $Vn \cap Vt = \emptyset$, P is the set of productions of the form $A \rightarrow w$ where $A \in Vn$ and $W \in V$ ($V_n \cup V_T = V$) and S is the axiom or start symbol. The relation $\Rightarrow$ and $\overset{*}{\Rightarrow}$ is defined as for regular grammars (cfr. def. 1.5.).

Example 2.1.

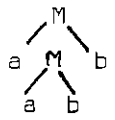
The grammar equivalent with the recursive t.n. M is G = ⟨Vn, Vt, P, S⟩ and $V_n = \{M\}$, $Vt = \{a,b\}$ and S = M, P:

1. M → a M b
2. M → ab

An example of a derivation:

$M \overset{1}{\Rightarrow} a\ M\ b \overset{2}{\Rightarrow} aabb$

or



G is not only a cfg., it can be proved that it is impossible to write a regular grammar generating $L(G) = a^n b^n$ because of the self embedding property of G.

To indicate the equivalence between cfg.'s and recursive t.n.'s we construct an algorithm to rewrite the one into the other and vice-versa.

1. From cfg. to recursive t.n.

Each nonterminal symbol A is a t.n. . Let $I1_A$ where A is a particular t.n., be the index

of the latest state of a t.n. during the construction process, in the beginning $I1_A = 1$. $I2_A$ is the index of the current state of a particular t.n. A. For every production of the grammar, the symbol (A) which is on the left of the production is the involved t.n.. In the beginning $I2_A$ is 1. For every right symbol in a production we try to make a transition in A from $I2_A$; if it is possible to do so $I2_A$ is the state after the transition, if not we make a new rule (k becomes k + 1) in the network where $r_{K,1}$ is the symbol in the production, $r_{K,2}$ is $A/I2_A$ $I1_A$ becomes $I1_A + 1$ and $r_{K,3} = A/I1_A$. $I2_A$ becomes $I2_A + 1$. If the symbol itself is a t.n. then the condition is this symbol followed by /1. If the symbol on the left is the last in the rule, we add ↑ to $r_{K,3}$ because it is a final state.

Example 2.2.

We take the grammar of example 2.1. the productions were:

1. S → a S b
2. S → ab

We start with production 1. The left symbol 'S' denotes the t.n. S. $I1_S = 1$ and $I2_S = 1$. Because there is not yet a t.n., we cannot scan 'a' therefore we make a new transition $I1_S = I1_S + 1$ from $S/I2_S$ to $S/I1_S$ that is from S/1 to S/2.

1.⟨a,S/1 ,S/2 ⟩

Now we take the second element in production 1. Again we can't scan the symbol and so make a new state: $I1_S = I1_S + 1$; because S ∈ Vn , we have:

2.⟨S/1 ,S/2 ,S/3 ⟩

Finally for this production the last symbol b and because b is the last symbol.

3.⟨b , S/3 , S/4↑⟩

Then we take the second production, the left symbol is again S, we try to scan the first symbol of 2 and it works $I2_S = 2$ (from $r_{K,3}$) and $I1_S$ remains 4.
Then we try to scan the second symbol in the right part of the production, namely 'b'. This time we can't make a transition, therefor a new state is made: $I1_S + 1$ and because b is the last symbol this state is a finel one:

4.⟨b , S/2 ,S/5↑ ⟩

The resulting t.n. is:

1.⟨a , S/1 , S/2 ⟩
2.⟨S/1, S/2, S/3↑ ⟩
3.⟨b , S/3, S/4↑ ⟩
4.⟨b , S/2, S/5 ⟩

2. From r.t.n. to cfg.

Every t.n. in the set is a nonterminal symbol and every symbol input for a transition that is not a t.n. is a terminal symbol.
For every t.n. A we make a production with at the left the nonterminal symbol A. For every input that appears in a path of consecutive states starting with A/1 and ending with A/i↑, a symbol at the right of the production is made.

If this input is a t.n. then we only use the name of a t.n. and not the index.

Example 2.4.

1.⟨a ,S/1 , S/2 ⟩

2.⟨S/1,S/2,S/3 ⟩

3.⟨b , S/3 , S/4↑ ⟩

4.⟨b , S/2 , S/5↑ ⟩

There is only one t.n. namely S, therefore Vn = {S}, the terminal symbols are {a, b};

We have two paths starting from S/1 and ending in S/i↑ :

rule (1), rule (2), rule (3)

rule (1), Rule (4)

Therefore we have 2 productions:

1. S → a S b

2. S → ab

3. PUSHDOWN AUTOMATA

Def. 2.3.

A pushdown automaton M (for short pda.), is a system $(K, \Sigma, \Gamma, \delta, q_0, Z_0, E)$ where

1. K is a finite set of states
2. Σ is a finite alphabet called the alphabet
3. Γ is a finite alphabet called the push down alphabet
4. q_0 is the initial state, $q_0 \in K$.
5. Z_0 in Γ is the start symbol, initially present on the pushdown store
6. $E \subseteq K$ is the set of final states
7. δ is a mapping from $K \times (\Sigma \cup \{\lambda\}) \times \Gamma$ to finite subsets of $K \times \Gamma^*$

$\delta(q,a,Z) = \{(p_1,\gamma_1), (p_2,\gamma_2), \ldots, (p_m,\gamma_m)\}$, q, p_i in K, a in Σ, Z in Γ en γ_i in Γ.

This means that if the system is in state q, having in the pushdown store Z and on the input tape appears the symbol a, then for any i, we can replace in the store Z by γ_i and the new state is p_i.

Example 2.3. (due to Hopcroft and Ullman (1969))

$M = (\{Q_1, Q_2\}, \{0, 1, C\}, \{R, B, G\}, \delta, q_1, R, \emptyset,)$

$\delta(Q_1, 0, R) = \{(Q_1, BR)\}$ $\quad$ $\delta(Q_2, 0, B) = \{(Q_2, \lambda)\}$

$\delta(Q_1, 0, B) = \{(Q_1, BB)\}$ $\quad$ $\delta(Q_2, \lambda, R) = \{(Q_2, \lambda)\}$

$\delta(Q_1, 0, G) = \{(Q_1, BG)\}$ $\quad$ $\delta(Q_1, 1, R) = \{(Q_1, GR)\}$

$\delta(Q_1, C, R) = \{(Q_2, R)\}$ $\quad$ $\delta(Q_1, 1, B) = \{(Q_1, GB)\}$

$\delta(Q_1, C, B) = \{(Q_2, B)\}$ $\quad$ $\delta(Q_1, 1, G) = \{(Q_1, GG)\}$

$\delta(Q_1, C, G) = \{(Q_2, G)\}$ $\quad$ $\delta(Q_2, 1, G) = \{(Q_2, \lambda)\}$

$L(M) = \{wcw^R \mid w \in \Sigma$ and $w^R\}$ denotes the reverse of w.

There is a theorem from formal language theory saying 'If L is a contextfree language, then there exists a pda M, such that L = N(M)' (Hopcroft & Ullman, theorem 5.2.). The proof is such that first a cfg. generating L is constructed, then this grammar is put into Greibach normal form and finally an algorithm is applied to extract a pda. from a cfg.

The important thing is that the grammar must be in Greibach normal form and although the weak generative capacity is not affected when rewriting an arbitrary cfg. into its corresponding Greibach normal form, the strong generative capacity (structural descriptions) certainly is. This is a serious drawback in natural language research where structural descriptions are at least as important as the language generated.

Although we will develop in the following paragraphs parsers for cfg.'s , recursive t.n.'s and pda's, the last sort of systems will not be of great interest to us. We will only mention the pda. parser for the sake of completeness.

4. TYPE 2 PARSERS

Definition 2.4. A parser $\mathcal{P} = (A, t_i, N, F)$ is called a type 2 parser if the underlying control structure or driver A is a context-free grammar, a recursive transition network or a pushdown automaton.

We give now examples of the three different parsers. We start with pda parsers because they are the easiest to understand. Then we approach recursive transition network parsers because they incorporate the basic ideas for a cfg. parser. Finally we construct a cfg. parser.

Example 2.4.

Let $\mathcal{P} = (A, t_i, N, F)$ be a parser where

(i) A is the pda. of example 2.3.

(ii) t_i is a triple $(a_{i,1}, a_{i,2}, a_{i,3})$

$a_{i,1}$ = the position to be read in the input string

$a_{i,2}$ = the state at this task

$a_{i,3}$ = the pushdown store

(iii) N:

1. the base: $t_1 = (1, Q_1, R)$ Q_1 is the start state and R is the initial configuration of the pushdown store . v = 1.

2. the recursion step:

(a) Let a rule in the pda. be $\delta(q,a,Z) = \{(p_1,\gamma_1), \ldots, (p_j,\gamma_j), \ldots, (p_m,\gamma_m)\}$. Then if $a = I(a_{i,1})$, $q = a_{i,2}$ and Z is the first sequence in $a_{i,3}$, we apply for every j in the rule, $1 \leq j \leq m$ (and v = v + 1)

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ p_j & \text{for } Y = a_{v,2} \\ \gamma_j k \quad \text{and } k = a_{i,3} - Z & \text{for } Y = a_{v,3} \end{cases}$$

(b) If at a certain stage $a_{i,3} = Z$ for a rule $\delta(q, \lambda, Z)$ then $(v = v + 1)$

$$(N(t_i))(Y) = \begin{cases} a_{i,1} & \text{for } Y = a_{v,1} \\ p_j & \text{for } Y = a_{v,2} \\ \gamma_i K \quad K = a_{i,3} - Z & \text{for } Y = a_{v,3} \end{cases}$$

3; the restriction

t_v is defined iff. $a_{i,1} + 1 \leq |\sigma| + 1$, iff $I(a_{i,1}) \in \Sigma$ and iff $a_{i,3} \neq \lambda$

(iv) F:

$\Theta \in 0,1$ if there is a task in T where $a_{i,1} = |\sigma| + 1$, $a_{i,3} = \lambda$, Θ is 0(means accepted), else $\Theta = 1$.

Example 2.4.1.

σ = 101c101 $\quad |\sigma| = 7$

$t_1 = \langle 1, Q_1, R \rangle$
$t_2 = \langle 2, Q_1, GR \rangle$
$t_3 = \langle 3, Q_1, BGR \rangle$
$t_4 = \langle 4, Q_1, GBGR \rangle$
$t_5 = \langle 5, Q_2, GBGR \rangle$
$t_6 = \langle 6, Q_2, BGR \rangle$
$t_7 = \langle 7, Q_2, GR \rangle$
$t_8 = \langle 8, Q_2, R \rangle$
$t_9 = \langle 8, Q_2, \lambda \rangle$

$\Theta = 0$ and $\sigma \in L(A)$

Problems 2.1.

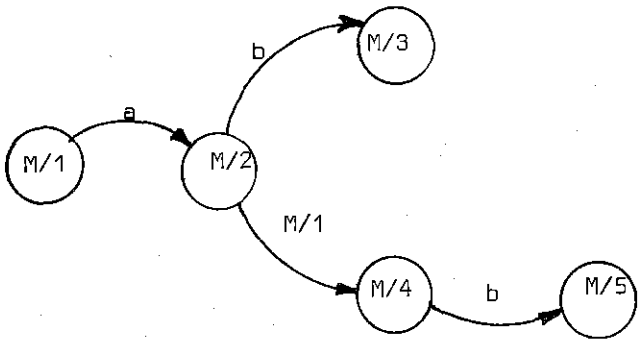
(i) Although the pda. A in example 2.4. was deterministic, the parser is nondeterministic. How can it be made deterministic?

(ii) Construct a nondeterministic pda. accepting $\{ww \mid w \in \{0,1\}^*\}$ and parse the words 001100, 110011.

Example 2.5.

Let $\mathcal{T} = \langle A, t_i, N, F \rangle$ be a recursive transition network parser and

(i) A is the following network:



in tabular form:

1. ⟨a, M/1 , M/2 ⟩
2. ⟨b, M/2 , M/3↑⟩
3. ⟨M/1, M/2, M/4⟩
4. ⟨b, M/4 , M/5↑⟩

(ii) t_i is a 6-tuple: $\langle a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4}, a_{i,5}, a_{i,6} \rangle$

$a_{i,1}$ = the position to be read on the input tape I

$a_{i,2}$ = the state of the automaton at the execution of a task

$a_{i,3}$ = either 1, 0, 2

$a_{i,4}$ = the anchor

$a_{i,5}$ = the index of the rule in the network used to construct this task.

$a_{i,6}$ = the index of a task were recursively another network was called.

(iii) N :

1. the base: $t_1 = \langle 1, M/1, 1, 0, 0, 1 \rangle$ $\quad (v = 1)$
2. recursive step:

(a) if $a_{i,3} = 1$ then for every rule with index k in the network where $a_{i,2} = r_{k,2}$ a new task is made (v = v +1) where

$$(N(t_i))(Y) = \begin{cases} a_{i,1} & \text{for } Y = a_{v,1} \\ r_{k,1} & \text{for } Y = a_{v,2} \\ \begin{cases} 1 & \text{if } r_{k,1} \text{ is a start state,} \\ 0 & \text{otherwise} \end{cases} & \text{for } Y = a_{v,3} \\ i & \text{for } Y = a_{v,4} \\ k & \text{for } Y = a_{v,5} \\ \begin{cases} v & \text{if } r_{k,1} \text{ is a start state} \\ a_{i,6} & \text{otherwise} \end{cases} & \text{for } Y = a_{v,6} \end{cases}$$

(b) if $a_{i,3} = 0$ then $(v = v + 1)$

(restriction if $I(a_{i,1}) \neq a_{i,2}$, t_v is undefined)

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ r_{a_{i,5},3} & \text{for } Y = a_{v,2} \\ \begin{cases} 2 \text{ if } r_{a_{i,5},3} \in E \text{ (set of final states)} \\ \text{else } 1. \end{cases} & \text{for } Y = a_{v,3} \\ i & \text{for } Y = a_{v,4} \\ a_{i,5} & \text{for } Y = a_{v,5} \\ a_{i,6} & \text{for } Y = a_{v,6} \end{cases}$$

(c) if $a_{i,3} = 2$ then $(v = v + 1)$

(restriction: if $a_{a_{i,6},5} = 0$ t_v is undefined

$$(N(t_i))(Y) \begin{cases} a_{i,1} & \text{for } Y = a_{v,1} \\ r_{a_{a_{i,6},5},3} & \text{for } Y = a_{v,2} \\ \begin{cases} 2 \text{ if } r_{a_{a_{i,6},5},3} \in E \text{ (set of final states)} \\ \text{else } 1 \end{cases} & \text{for } Y = a_{v,3} \\ i & \text{for } Y = a_{v,4} \\ a_{a_{i,6},5} & \text{for } Y = a_{v,5} \\ a_{a_{a_{i,6},4},6} & \text{for } Y = a_{v,6} \end{cases}$$

(iv) F:

We define a valid start task in σ as a task t_i such that $a_{i,1} = 1$, $a_{i,2}$ = the start state and $a_{i,5} = 0$

We define a valid end task in σ as a task t_i such that $a_{i,1} = |\sigma| + 1$, $a_{i,2} \in E$ and $a_{i,6} = 1$

We define a valid path P through T (the set of tasks) as a sequence of indices where

(i) i_1 is the index of a valid end task in T

(ii) $i_{j+1} = a_{i_j,4}$

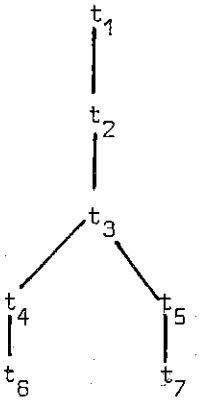
(iii) if $a_{i_j,4} = 0$ then P is complete.

$\Theta = 0$ if there is a valid path P through T else $\Theta = 1$

Example 2.5.1.

σ = ab

t_1 = ⟨ 1, M/1, 1, 0, 0, 1 ⟩
t_2 = ⟨ 1, a , 0, 1, 1, 1 ⟩
t_3 = ⟨ 2, M/2, 1, 2, 1, 1 ⟩
t_4 = ⟨ 2, b , 0, 3, 2, 1 ⟩
t_5 = ⟨ 2, M/1, 1, 3, 3, 5 ⟩
t_6 = ⟨ 3, M/3↑, 2, 4, 2, 1 ⟩
t_7 = ⟨ 2, a , 0, 5, 1, 5 ⟩



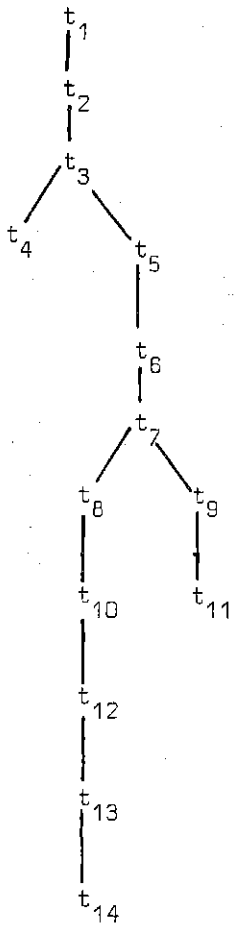
Θ = 0 because t_6 is a valid end task . $P_{t_{16}}$ = ⟨ 6, 4, 3, 2, 1 ⟩

Example 2.5.2.

σ = aabb

t_1 = ⟨ 1,M/1 , 1, 0, 0, 1 ⟩
t_2 = ⟨ 1,a , 0, 1, 1, 1 ⟩
t_3 = ⟨ 2,M/2 , 1, 2, 1, 1 ⟩
t_4 = ⟨ 2,b , 0, 3, 2, 1 ⟩
t_5 = ⟨ 2,M/1, 1, 3, 3, 5 ⟩
t_6 = ⟨ 2,a , 0 , 5, 1, 5 ⟩
t_7 = ⟨ 3, M/2, 1, 6, 1, 5 ⟩
t_8 = ⟨ 3, b , 0 , 7, 2, 5 ⟩
t_9 = ⟨ 3, M/1, 1, 7, 3, 9 ⟩
t_{10} = ⟨ 4,M/3↑,2 , 8, 2, 5 ⟩
t_{11} = ⟨ 3, a , 0, 9, 1, 9 ⟩
t_{12} = ⟨ 4,M/4 , 1, 10,3, 1 ⟩
t_{13} = ⟨ 4, b , 0, 12,4 ,1 ⟩
t_{14} = ⟨ 4,M/5↑, 2, 13,4, 1 ⟩

Θ = 0 because t_{14} is a valid end task. The path:
$P_{t_{14}}$ = ⟨14,13,12,10,8,7,6,5,3,2,1⟩



Now we construct a context-freegrammar parser, the idea is to use a recursive transition network parser but with the ideas of the algorithm to rewrite cfg. into t.n.'s incorporated in N .

Example 2.6.

Let $\mathcal{S}$ = ⟨A, t_i, N, F⟩ be a context-free grammar parser and

(i) is a cfg. ⟨ Vn, Vt, P, S ⟩ where Vn = {S} , Vt = {a,b} and P:

1. S → ab
2. S → a S b

(ii) t_i is a 7-tuple: ⟨ $a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4}, a_{i,5}, a_{i,6}, a_{i,7}$ ⟩

$a_{i,1}$ = the position to be read on the input tape

$a_{i,2}$ = the symbol of V where the task is interested in

$a_{i,3}$ = the position in a production after $a_{i,2}$ and the first symbol on the right is in position 1

$a_{i,4}$ = either 1, 0, or 2

$a_{i,5}$ = the anchor

$a_{i,6}$ = the index of a production consulted to construct this task

$a_{i,7}$ = the index of the task where the production was first consulted.

(iii) N:

1. the base: t_1 = ⟨ 1, S, 1, 1, 0, 0, 1 ⟩ v= 1

2. the recursive step:

(a) If $a_{i,4}$ = 1, a new production is consulted, therefore for every production j where $a_{i,2}$ appears on the left: (v = v +1)

$$[N(t_i)](Y) = \begin{cases} a_{i,1} & \text{for } Y = a_{v,1} \\ \text{the first symbol on the right of } j & \text{for } Y = a_{v,2} \\ 2 & \text{for } Y = a_{v,3} \\ \begin{cases} 1 \text{ if } a_{v,2} \in Vn \\ 0 \text{ otherwise} \end{cases} & \text{for } Y = a_{v,4} \\ i & \text{for } Y = a_{v,5} \\ j & \text{for } Y = a_{v,6} \\ \begin{cases} v & \text{if } a_{v,2} \in Vn \\ a_{i,6} & \text{otherwise} \end{cases} & \text{for } Y = a_{v,7} \end{cases}$$

(b) if $a_{i,4}$ = 0 then (if $I(a_{i,1}) = a_{i,2}$ else t_v is undefined)

(i) if the $a_{i,3}$ symbol on the right of $a_{i,5}$ is not empty

$$
\begin{cases}
a_{i,1} + 1 & \text{for } Y = a_{v,1} \\
\text{the } a_{i,3} \text{ symbol on the right of } a_{i,5} & \text{for } Y = a_{v,2} \\
a_{i,3} + 1 & \text{for } Y = a_{v,3} \\
\begin{cases} 1 \text{ if } a_{v,2} \in Vn \\ 0 \end{cases} & \text{for } Y = a_{v,4} \\
i & \text{for } Y = a_{v,5} \\
j & \text{for } Y = a_{v,6} \\
\begin{cases} v \text{ if } a_{v,2} \in Vn \\ \text{else } a_{i,6} \end{cases} & \text{for } Y = a_{v,7}
\end{cases}
$$

(ii) else:

Restriction: if $a_{i,7} = 1$ and $a_{i,1} = (\sigma)$, t_i is a valid end task, if $a_{i,7} = 1$, t_v is undefined, else

$$
\begin{cases}
a_{i,1} & \text{for } Y = a_{v,1} \\
a_{a_{i,7},5} \text{ symbol on the right of the production specified in } a_{a_{i,7},6} & \text{for } Y = a_{v,2} \\
a_{a_{i,7},3} + 1 & \text{for } Y = a_{v,3} \\
\begin{cases} 1 \text{ if } a_{v,2} \in Vn \\ 0 \text{ otherwise} \end{cases} & \text{for } Y = a_{v,4} \\
i & \text{for } Y = a_{v,5} \\
a_{a_{i,7},6} & \text{for } Y = a_{v,6} \\
a_{a_{a_{i,7},5},7} & \text{for } Y = a_{v,7}
\end{cases}
$$

(iv) $F: T \to \Theta$ is such that $\Theta = 0$ if there is a valid end task in T, else $\Theta = 1$.

Example 2.6.1.

$\sigma = ab$

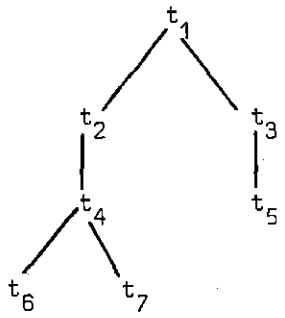
$t_1 = (1, S, 1, 1, 0, 0, 1)$

$t_2 = (1, a, 2, 0, 1, 1, 1)$

$t_3 = (1, a, 2, 0, 1, 2, 1)$

t_4 = ⟨ 2, b, 3, 0, 2, 1, 1 ⟩
t_5 = ⟨ 2, S, 3, 1, 3, 2, 5 ⟩
t_6 = ⟨ 2, a, 2, 0, 5, 1, 5 ⟩
t_7 = ⟨ 2, a, 2, 0, 5, 2, 5 ⟩

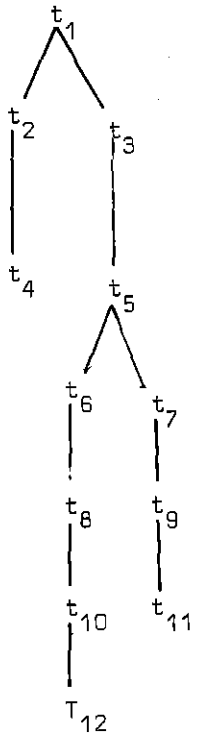
t_4 is a valid end task and Θ= 0



Example 1.6.2.

σ = aabb

t = ⟨ 1, S, 1, 1, 0, 0, 1 ⟩
t = ⟨ 1, a, 2, 0, 1, 1, 1 ⟩
t = ⟨ 1, a, 2, 0, 1, 2, 1 ⟩
t = ⟨ 2, b, 3, 0, 2, 1, 1 ⟩
t = ⟨ 2, S, 3, 1, 3, 2, 5 ⟩
t = ⟨ 2, a, 2, 0, 5, 1, 5 ⟩
t = ⟨ 2, a, 2, 0, 5, 2, 5 ⟩
t = ⟨ 3, b, 3, 0, 6, 1, 5 ⟩
t = ⟨ 3, S, 3, 1, 7, 2, 9 ⟩
t = ⟨ 4, b, 4, 0, 8, 2, 1 ⟩
t = ⟨ 3, a, 2, 0, 9, 1, 9 ⟩
t = ⟨ 3, a, 2, 0, 9, 2, 9 ⟩



t_{10} is a valid end task and Θ = 0

5. SOME APPLICATIONS OF THE FILTER

It would lead us too far to develop here in detail interesting applications. The reader is referred to Steels (1974) for algorithms to extract strings with labeled bracketings and structural descriptions in the form of trees from the set of tasks.

It is also possible to define actions (e.g. operations over trees) in the filter, the result is then so called augmented transition networks.

6. SOME DESIGN REMARKS

The type 2 parsers were all what is usually called top down and breadth first. That this must not necessarily be so may be illustrated by the following ideas of which the formal definition is left to the reader.

(a) Top down with back track.
This can be done by using the same type of parser with the difference that each time not all tasks resulting from N are executed but only one of them. N keeps track of the structural description of the parsing process and if a certain task is undefined, then it returns to the node (i.e. the task) of the parsing tree where there was another task resulting from the execution of N. This task is then further executed and so on.

(b) Bottom up.
The notion of a task can also be applied to bottom up parsers, in this case tasks carry the information from where to where a constituent goes, what the name of the constituent is, how it was formed, etc.. . Each task then creates a new higher constituent if this is possible or proceeds in the inputstring. If at the end there is a task with a constituent equal to the axiom or start symbol and ranging over the whole input, then the word is accepted.

There are many other ways to design parsers (e.g. in mixed mode), but we think that the basic mechanism, the creation of tasks by functions, will always remain.

7. HISTORICAL NOTES

These notes are by no means exhaustive and only cover applications for natural language parsers. Note also that in the paper we only treated fundamentals of parsers, how they are worked out in practice is even a more complex matter, e.g. things are added such as probability of likelihood for a path, etc... .

For another approach to the problem, we refer to Aho & Ullman (1973) . Transition networks and their equivalents are used for morphographemic rewriting or orthographic decoding(cf. Kay (1974))

Also it is possible to consider a dictionary as a f.s machine and then consulting the dictionary becomes again a type 3 parser. If the language accepted is the product with itself, then the parser can find all possible parts in a word. This is an alternative for the Reiffler calculus (used e.g. in Verloren van Themaat (1972)).

Transition networks are also a basic data structure in the MIND system (cf. Kay, 1973) ('The chart' is interpreted as a transition network', 161) and the General syntactic processor (Kaplan (1973)). The notion of parsing with tasks is found in Kay (1974) and some ALGOL 60 programs are given there.

Recursive transition networks or basic transition networks (BTN) were introduced by Woods (1970) and extended to augmented transition networks by the addition to the model of arbitrary register-setting actions and arbitrary conditions on the arcs' (Woods 1973, 116). They are used in many syntactic and morphological analysers: (eg; Woods (1973), Simmons (1974)). Also here we find the basic ideas of the parsers as was developed in this paper. Woods describes his implementation as follows: 'the most natural way of thinking of its operation is in terms of the notions of instantaneous machine configuration' (i.e. tasks), and transition functions (a function which computes successors to given instantaneous configurations).'

In some earlier parsers (cfr the predictive analyzer, Kuno and Oettinger (1965) and the selective top-to-bottom Algorithm by Griffiths and Petrick (1965) the concept of pushdown automata was used with the result that the grammars must be in some normal form. See eg. Kuno (1967) for a discussion and solution of these problems;

There have been alternative ways of parsing natural language on lower levels e.g. PROGRAMMAR (Winograd 1972), which is a language to write parsers in, some parsers with limited dictionaries and search strategies based on word order to back upthis lack of information (cf. Thorne (1970)), parsers resulting in distributional analysis (of Salkoff (1973)), etc... We think however that most of them are too much biased by the particular grammar or language. Indeed they all have in common that the grammar (control structure) is not input to the parser but incorporated into the format of the parser.

It is also our experience that a task-oriented parser is very interesting when designing larger systems where different subsystems (i.e. parsers)all interact.

3. Bibliography

Aho, A.V. and J.D. Ullman:

(1973) The theory of parsing, translation and compiling. Vol I. Parsing.Prentice Hall, Englewood Cliffs, New Jersey.

Chomsky, N.:

(1963) On certain formal properties of grammars. In: Luce, R. et.al. (ed) Handbook of Mathematical Psychology, Vol II. Wiley, New York.

Griffiths, T. and S.R. Petrick:

(1965) On the relative efficiencies of context-free grammar recognizers, Comm. Assoc. Comput. Mach. 12, no 1, 42 - 52.

Hopcroft, J.E. and J.D. Ullman:

(1969) Formal languages and their relation to automata. Addison-Wesley Publishing Company, London.

Kay, M.:

(1973) The mind system. In: Rustin (1973).

Kuno, S.:

(1967) Computer analysis of natural languages. Proceedings of Symposia in applied mathematics, 52-110. Mathematical aspects of computer science. American Mathematical Society, Providence, Rhode Island.

Kuno, S. and A.G. Oettinger:

(1974) Vraagstukken omtrent de representatie van linguistische informatie. UIA. Technisch rapport. December 1974.

Minsky, M.:

(1967) Computation. Finite and infinite machines. Prentice Hall, Englewood Cliffs, New Jersey.

Rustin, R.:

(1973) Natural language processing. Algorithmics press, New York.

Salkoff, M.

(1973) Une grammaire en châine du Français. Analyse distributionelle. Dunod, Paris.

Saloma, A.:

(1973) Formal languages. Academic Press, New York.

Simmons, R.F.:

(1973) An experimental parsing system for transition network grammars. In: Rustin (1973).

Woods, W.:

(1970) Transition network grammars for natural language analysis. Communications of the ACM, Vol 13, Dec 1970.

Winograd, T.:

(1972) Understanding natural language. Academic Press, New York and London.

Verloren van Themaat, W.A.:

(1973) A general syntactic processor. In: Rustin (1973).

Kaplan, R.:

(1972) Automatic analysis of dutch compound words. Mathematical Centre Tracts 38. Mathematisch centrum Amsterdam.

Thorne, J.P., et.al.:

(1970) A model for syntactic description. Sci. Rep. English Language Research Unit, University of Edinburgh, Edinburgh.

APPENDIX A.

(Solutions to problems)

1.1.

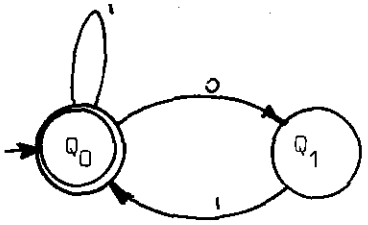
(i) a. $Q_0 \xrightarrow{1} Q_1 \xrightarrow{0} Q_2 \xrightarrow{1} Q_3 \xrightarrow{0} Q_0 \xrightarrow{0} Q_3$ ('10100' is not accepted)

b. $Q_0 \xrightarrow{0} Q_3 \xrightarrow{1} Q_2 \xrightarrow{0} Q_1$ ('010' is not accepted)

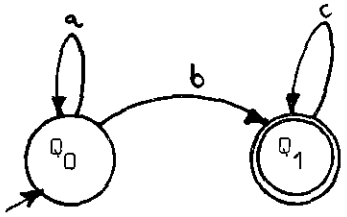
c. $Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_0 \xrightarrow{1} Q_1 \xrightarrow{0} Q_2 \xrightarrow{0} Q_1 \xrightarrow{0} Q_2$ (' 000' is accepted)

(ii) The t.n. is the same

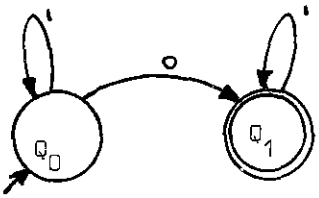
(iii)



(iv)



(v)



1.2.

(i) The required grammar is $G = \langle Vn, Vt, P, A \rangle$, a regular grammar, and $Vn = \{A,B,C\}$, $Vt = \{a, b, c\}$ and P:

1. A → c A
2. A → a A
3. A → a B
4. B → b C
5. C → a A
6. A → c
7. A → a
8. C → a

Let $\mathcal{S}$ = (G, t_i, N, F) be a regular grammar parser and $\mathcal{S}$ is the same as in example 1.13. except for F.

Let θ_i = ($\beta_{i,1}$, $\beta_{i,2}$) and $\beta_{i,1}$ is a terminal symbol and $\beta_{i,2}$ is a nonterminal symbol. θ_1 = ⟨λ, A⟩

Let P be a valid path through T, then we construct as follows:

For very element j starting from the second last one and going to the first one in a path P, we construct a pair ⟨ $\beta_{i,1}$, $\beta_{i,2}$ ⟩ and $\beta_{i,1}$ = I($\alpha_{j,1}$ - 1), $\beta_{i,2}$ = $\alpha_{j,2}$

(i) σ = 'c'

t_1 = ⟨ 1, A, 0, 0 ⟩

t_2 = ⟨ 2, A, 1, 2 ⟩

t_3 = ⟨ 2, λ, 1, 7 ⟩

t_4 = ⟨ 2, B, 1, 3 ⟩

t_3 is a valid end task and $P_{t_{13}}$ = ⟨ 3, 1 ⟩

Ø: θ_1 = ⟨λ, A⟩

θ_2 = ⟨ c, λ ⟩

(ii) σ = bac

t_1 = ⟨1, A, 0, 0⟩

(the parser blocks because there is no production as required in the recursive step) is not accepted by G.

(iii) σ = cabacccabaa

t_1 = ⟨ 1, A, 0, 0 ⟩

t_2 = ⟨ 2, A, 1, 1 ⟩

t_3 = ⟨ 3, A, 2, 2 ⟩

t_4 = ⟨ 3, λ, 2, 7 ⟩

t_5 = ⟨ 3, B, 2, 3 ⟩

t_6 = ⟨ 4, C, 5, 4 ⟩

t_7 = ⟨ 5, A, 6, 5 ⟩

t_8 = ⟨ 6, A, 7, 1 ⟩

t_9 = ⟨ 7, A, 8, 1 ⟩

t_{10} = ⟨ 8, A, 9, 1 ⟩

t_{11} = ⟨ 9, A, 10, 2 ⟩

t_{12} = ⟨ 9, B, 10, 3 ⟩

t_{13} = ⟨ 9, λ, 10, 7 ⟩

t_{14} = ⟨10, C, 12, 4 ⟩

t_{15} = ⟨11, A, 14, 5 ⟩

t_{16} ⟨11, λ, 14, 8 ⟩

t_{17} = ⟨12, A, 15, 2 ⟩

t_{18} = ⟨12, B, 15, 3 ⟩

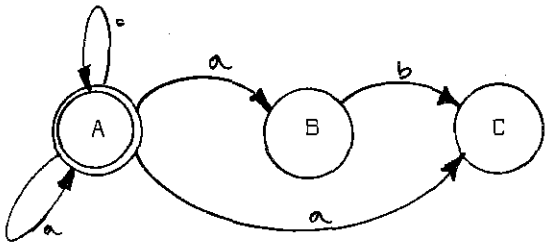
t_{19} = ⟨12, λ, 15, 7 ⟩

T_{19} is a valid end task and $P_{t_{19}}$ is $\langle 19, 15, 14, 12, 10, 9, 8, 7, 6, 5, 2, 1 \rangle$

$\theta_1 = \langle \lambda, A \rangle$
$\theta_2 = \langle c, A \rangle$
$\theta_3 = \langle a, B \rangle$
$\theta_4 = \langle b, C \rangle$
$\theta_5 = \langle a, A \rangle$
$\theta_6 = \langle c, A \rangle$
$\theta_7 = \langle c, A \rangle$
$\theta_8 = \langle c, A \rangle$
$\theta_9 = \langle a, B \rangle$
$\theta_{10} = \langle b, C \rangle$
$\theta_{11} = \langle a, A \rangle$
$\theta_{12} = \langle a, \lambda \rangle$

As one can see Θ is in fact a description of the tree which corresponds to the generation of the string.

(ii) The parser is equivalent to the regular grammar parser of example 1.13. except that the t.n. is A:



and the recursion step goes as follows:

For every rule $r_{k,j}$ is equal to $I(a_{i,1})$ and $r_{k,2}$ is $a_{i,2}$

$$(N(t_i))(Y) = \begin{cases} a_{i,1} + 1 & \text{for } Y = a_{v,1} \\ r_{k,3} & \text{for } Y = a_{v,2} \\ i & \text{for } Y = a_{v,3} \\ k & \text{for } Y = a_{v,4} \end{cases}$$

(iii) The parser is the same as in example 1.13. except for F.

Let $\Theta = 0$ if σ is ambiguous, else $\Theta = 1$. $F: T \rightarrow \Theta$ is such that if there is more then one valid end task in T, $\Theta = 0$, else $\Theta = 1$

2.1.

(i) if the pda. is deterministic, then j in the recursive step is always equal to 1. Hence we change the condition in 2(a):

'If (...) we apply for the pair on the left side of the rule (...)'.

(ii) The pda is:

$M = \langle \{Q_1, Q_2\}, \{0,1\}, \{R,B,G\}, \delta, Q_1, R, \emptyset \rangle$

1. $\delta(Q_1, 0, R) = \{(Q_1, BR)\}$
2. $\delta(Q_1, 1, R) = \{(Q_1, GR)\}$
3. $\delta(Q_1, 0, B) = \{(Q_1, BB), (Q_2, \lambda)\}$
4. $\delta(Q_1, 0, G) = \{(Q_1, BG)\}$
5. $\delta(Q_1, 1, B) = \{(Q_1, GB)\}$
6. $\delta(Q_1, 1, G) = \{(Q_1, GG), (Q_2, \lambda)\}$
7. $\delta(Q_2, 0, B) = \{(Q_2, \lambda)\}$
8. $\delta(Q_2, 1, G) = \{(Q_2, \lambda)\}$
9. $\delta(Q_1, \lambda, R) = \{(Q_2, \lambda)\}$
10. $\delta(Q_2, \lambda, R) = \{(Q_2, \lambda)\}$